Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE- SEMESTER–IV (NEW) EXAMINATION – WINTER 2024

Subject Code:3140702 Date:19-11-2024

Subject Name: Operating System

Time:02:30 PM TO 05:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1	**(a)**	What is Operating System? Explain basic functions of OS.	**03**
	(b)	Explain Virtual Machine architecture of OS.	**04**
	(c)	Differentiate Multiprogramming, Multitasking, Multiprocessing OS.	**07**
Q.2	**(a)**	Explain Advantages of using threads in Process.	**03**
	(b)	Explain how System call is handled in OS with example.	**04**
	(c)	Differentiate process and thread. Explain process state diagram.	**07**
		OR	
	(c)	Explain Peterson's Solution to achieve mutual exclusion.	**07**
Q.3	**(a)**	Differentiate Pre-emptive and Non-Preemptive process scheduling.	**03**
	(b)	Explain the difference between internal and external fragmentation.	**04**
	(c)	What is semaphore? Solve the producer consumer problem with Semaphore.	**07**
		OR	
Q.3	**(a)**	What is deadlock? List the conditions that lead to deadlock.	**03**
	(b)	Explain Monitor construct to achieve mutual exclusion.	**04**
	(c)	Solve the Dining philosopher problem with semaphore.	**07**
Q.4	**(a)**	Explain the difference between logical and physical addresses.	**03**
	(b)	What is TLB(Translation Lookaside Buffer), and what is use of it?	**04**
	(c)	Explain following process scheduling algorithms. i) First Come First Serve ii) Non-preemptive Shortest Job First iii) Shortest Remaining Time Next iv) Round Robin	**07**
		OR	
Q.4	**(a)**	What is virtualization? Explain the benefits of virtualization.	**03**
	(b)	Define Virtual Memory. Explain the process of converting virtual addresses to physical addresses with a neat diagram.	**04**
	(c)	Consider a Swapping system in which memory consists of the following hole sizes(in MB) in memory order: 10, 4,20,18,7,9,12,15. Which hole is taken for successive segment request of 12 MB and 10 MB for First fit, Best fit, Worst fit and Next fit.	**07**
Q.5	**(a)**	What is Inode? Explain it's usage.	**03**
	(b)	List Advantages of LINUX/UNIX operating system over Windows.	**04**
	(c)	Explain various Page Replacement Algorithms.	**07**

OR

Q.5	**(a)**	Explain I/O buffering in brief.	**03**
	(b)	Explain Disk arm scheduling algorithms.	**04**
	(c)	Explain the Banker's algorithm for deadlock avoidance with an example.	**07**

****

Q.1 (a) What is Operating System? Explain basic functions of OS.

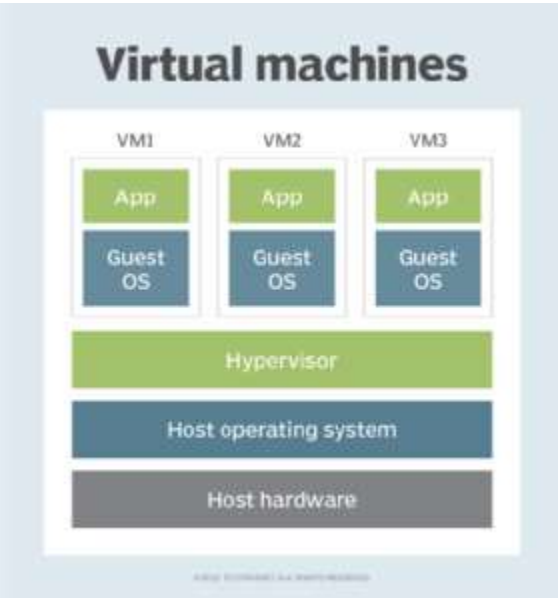
Operating System

An Operating System (OS) is a system software that controls the overall working of a computer system. It acts as an interface between the user and the computer hardware. The operating system manages all hardware resources such as CPU, memory, storage devices, and input/output devices. It provides a platform on which application programs can run smoothly and efficiently. Without an operating system, a computer system cannot function properly.

Basic Functions of Operating System

1. **Process Management**
   The operating system is responsible for creating, scheduling, and terminating processes. It decides which process should get CPU time and for how long. The OS also handles synchronization and communication between processes to ensure proper execution.
2. **Memory Management**
   The OS manages the main memory of the system. It keeps track of which memory is being used and which is free. The operating system allocates memory to programs when they start execution and deallocates memory when they finish.
3. **File Management**
   The operating system manages files stored on secondary storage devices. It provides facilities to create, read, write, delete, and organize files and directories. It also maintains file access permissions to protect data.

(b) Explain Virtual Machine architecture of OS.



Virtual Machine architecture allows multiple operating systems to run on a single physical computer system. In this architecture, the physical hardware is abstracted and shared among

different virtual machines. Each virtual machine behaves like an independent computer system with its own operating system, applications, and resources.

The key component of virtual machine architecture is the **Virtual Machine Monitor (VMM)**, also known as the **Hypervisor**. The hypervisor sits between the hardware and the guest operating systems. It controls and manages the allocation of hardware resources such as CPU, memory, and I/O devices to each virtual machine.

1. **Host Hardware**
   This is the physical hardware of the computer system. It includes CPU, main memory (RAM), hard disk, network cards, and input/output devices. All virtual machines ultimately use these physical resources.

2. **Host Operating System**
   The host operating system runs directly on the physical hardware. It is responsible for basic hardware control and provides a platform on which the hypervisor operates. Examples: Windows, Linux, macOS.

3. **Hypervisor (Virtual Machine Monitor)**
   The hypervisor is the most important component of virtual machine architecture. It creates, manages, and controls multiple virtual machines. It allocates CPU time, memory, and I/O resources to each VM and ensures isolation between them so that one VM does not affect others.

4. **Virtual Machines (VM1, VM2, VM3)**
   Each virtual machine acts like an independent computer system. Every VM has:

- Its own **Guest Operating System**
- Its own **Applications**
- Virtual CPU, memory, and devices

5. **Guest Operating System**
   The guest OS runs inside a virtual machine. It may be different from the host OS and other guest OSs. Each guest OS believes it has full control over hardware, but actually the hypervisor manages access.

6. **Applications**
   Applications run on top of the guest operating system just like in a normal computer system. Applications inside one VM are completely isolated from applications in other VMs.

Key Points

- Multiple operating systems can run on one physical system
- Hardware resources are efficiently shared
- Strong isolation between virtual machines
- Useful for testing, server consolidation, and cloud computing

Advantages of Virtual Machine Architecture

- Efficient utilization of hardware resources
- Isolation between operating systems
- Easy testing and development environment
- Better system security and flexibility

(c) Differentiate Multiprogramming, Multitasking, Multiprocessing OS.

Parameter	Multiprogramming OS	Multitasking OS	Multiprocessing OS
Basic Meaning	Multiprogramming is an operating system technique in which multiple programs are kept in main memory and the CPU executes one program at a time.	Multitasking is an operating system technique where a single user can execute multiple tasks simultaneously.	Multiprocessing is an operating system technique that uses more than one CPU to execute processes simultaneously.
Number of CPUs Used	Uses a single CPU.	Uses a single CPU.	Uses two or more CPUs.
Execution Method	CPU switches to another program when the current program is waiting for I/O.	CPU time is divided into small time slices and shared among tasks.	Multiple CPUs execute multiple processes at the same time.
Main Objective	To improve CPU utilization.	To improve user interaction and responsiveness.	To increase system performance and throughput.
User Interaction	Very little or no user interaction.	High user interaction.	Depends on system usage.
Parallel Execution	No true parallel execution.	No true parallel execution, only time sharing.	True parallel execution is possible.
Response Time	High response time.	Low response time.	Very low response time.
System Complexity	Simple to implement.	Moderately complex.	Highly complex.
Cost	Low cost.	Moderate cost.	High cost due to multiple processors.

Parameter	Multiprogramming OS	Multitasking OS	Multiprocessing OS
Examples	Batch operating systems.	Desktop operating systems like Windows, Linux.	Server systems, supercomputers.
Throughput	Moderate throughput.	Good throughput.	Very high throughput.
Fault Tolerance	Low fault tolerance.	Moderate fault tolerance.	High fault tolerance (one CPU can fail without stopping system).

Q.2 (a) Explain Advantages of using threads in Process.

1. **Improved Responsiveness**
   Threads allow a process to continue execution even if one thread is blocked due to I/O operations.

   This improves the responsiveness of applications, especially in interactive and real-time systems.

2. **Better CPU Utilization**
   Multiple threads can run concurrently within a single process.

   While one thread waits for I/O, another thread can use the CPU, leading to efficient utilization of processor time.

3. **Faster Context Switching**
   Context switching between threads of the same process is faster than switching between processes because threads share the same address space and resources.

4. **Resource Sharing**
   Threads within a process share memory, files, and other resources.

   This reduces overhead and makes communication between threads easier and faster.

5. **Reduced Memory Overhead**

   Threads require less memory compared to processes since they share code and data segments of the parent process.

6. **Improved Performance**
   Multithreaded applications can perform tasks in parallel, which improves overall system and application performance.

7. **Scalability on Multi-core Systems**
   Threads can run on different cores in multi-core processors, improving scalability and execution speed.

8. **Efficient Communication**
   Inter-thread communication is simpler and faster than inter-process communication because threads share the same memory space.

9. **Better Throughput**
   Using multiple threads allows more work to be completed in less time, increasing system throughput.

10. **Simpler Program Structure**
    Complex applications can be divided into smaller, manageable threads, making program design easier and more organized.

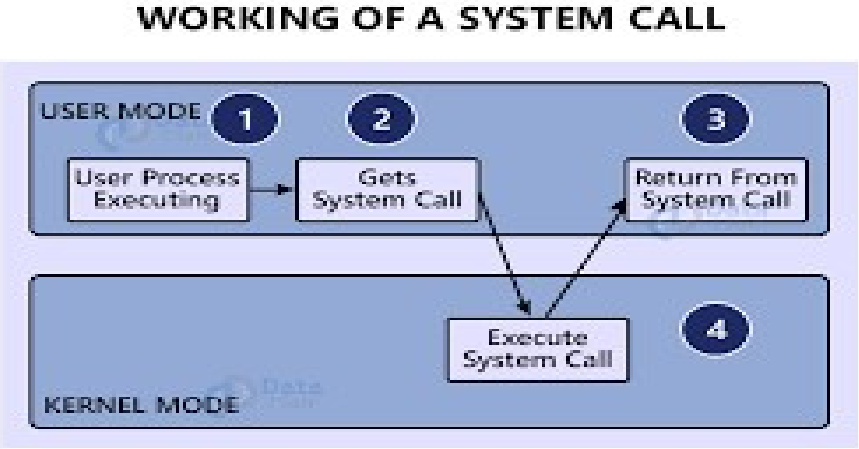
(b) Explain how System call is handled in OS with example.

A system call is a mechanism through which a user program requests a service from the operating system kernel. Since user programs run in user mode and cannot directly access hardware or critical system resources, system calls provide a controlled interface to interact with the OS.

When a user program needs a service such as file access or process creation, it first calls a library function. This library function places the system call number and required parameters in specific registers or memory locations. Then a special instruction such as **trap**, **syscall**, or **interrupt** is executed. This instruction switches the CPU from user mode to kernel mode.

After entering kernel mode, the operating system identifies the system call number, checks the validity of parameters, and invokes the corresponding kernel routine. The requested service is performed by the kernel. Once the operation is completed, the result is returned to the user program, and the CPU switches back to user mode.

Example:
When a program wants to read data from a file, it uses the `read()` system call. The user program calls the `read()` function, control is transferred to the kernel, the OS reads data from the file, and the data is returned to the program.



Working Flow of System Call Handling in Operating System

1. A user program running in **user mode** requests an operating system service such as file access, process creation, or device operation.
2. The user program invokes a **library function** (for example, `read()`, `write()`, `fork()`), which prepares the required parameters and system call number.
3. The library function executes a **system call instruction** (trap / interrupt / syscall), which transfers control from user mode to **kernel mode**.
4. The CPU switches to kernel mode and transfers control to the **system call handler** of the operating system.
5. The operating system identifies the requested system call using the system call number.
6. The OS validates parameters and checks permissions to ensure safe execution.
7. The corresponding **kernel routine** is executed to perform the requested service.
8. After completing the operation, the result or status is stored in registers or memory.
9. Control is returned to the user program, and the CPU switches back to **user mode**.
10. The user program continues execution from the point where the system call was made.

(c) Differentiate process and thread. Explain process state diagram.

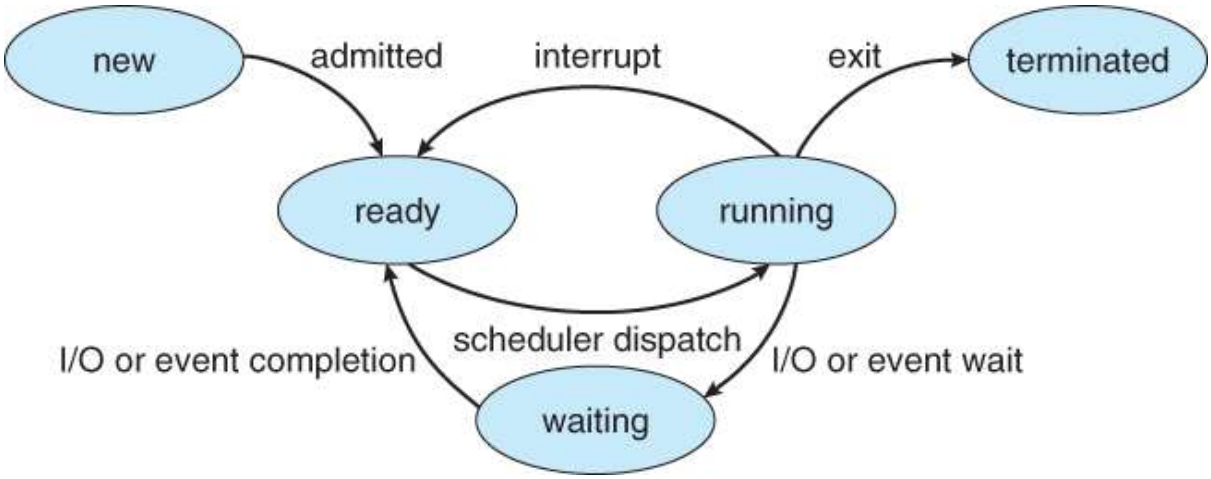
Difference between Process and Thread

Parameter	Process	Thread
Definition	A process is a program in execution along with its own resources.	A thread is the smallest unit of CPU execution within a process.
Address Space	Each process has its own separate address space.	Threads share the address space of the process.

Parameter	Process	Thread
Resource Ownership	Owns resources such as memory, files, and I/O devices.	Shares resources of the parent process.
Communication	Inter-process communication is complex and costly.	Inter-thread communication is simple and fast.
Context Switching	Context switching between processes is slow.	Context switching between threads is fast.
Creation Time	Process creation takes more time.	Thread creation takes less time.
Memory Requirement	Requires more memory.	Requires less memory.
Failure Impact	Failure of one process does not affect others.	Failure of a thread may affect the entire process.

Process State Diagram

A process state diagram represents different states through which a process passes during its execution. The operating system changes the state of a process based on CPU availability, I/O operations, and process completion.



States of a Process

1. **New State**

   When a process is created, it enters the new state. In this state, the operating system creates the process control block and allocates necessary resources.

2. **Ready State**

   After creation, the process enters the ready state. In this state, the process is fully prepared to execute and is waiting for CPU allocation.

3. **Running State**

   When the CPU scheduler selects a process from the ready queue, the process enters the running state. In this state, instructions of the process are executed by the CPU.

4. **Waiting (Blocked) State**

   A process enters the waiting state when it needs to wait for an event such as completion of I/O operation or availability of a resource.

5. **Terminated State**

   When a process completes execution or is terminated by the operating system, it enters the terminated state. All allocated resources are released.

OR (c) Explain Peterson's Solution to achieve mutual exclusion.

Peterson's Solution is a software-based solution used to achieve **mutual exclusion** between two processes that share common resources. It ensures that only one process enters the critical section at a time without using any special hardware instructions.

Peterson's solution is applicable only for **two processes**, generally named P0 and P1. It uses two shared variables: an integer variable `turn` and a boolean array `flag[2]`.

The variable `flag[i]` indicates whether process Pi wants to enter the critical section. The variable `turn` indicates whose turn it is to enter the critical section.

Working of Peterson's Solution

When a process wants to enter the critical section, it first sets its flag to true, indicating its intention to enter. Then it sets the `turn` variable to the other process, giving the other process a chance to enter first. After this, the process waits in a loop until the other process is either not interested in entering the critical section or it is its own turn.

If both processes try to enter the critical section at the same time, the `turn` variable ensures that only one process enters while the other waits. After completing the execution of the critical section, the process sets its flag to false, allowing the other process to enter.

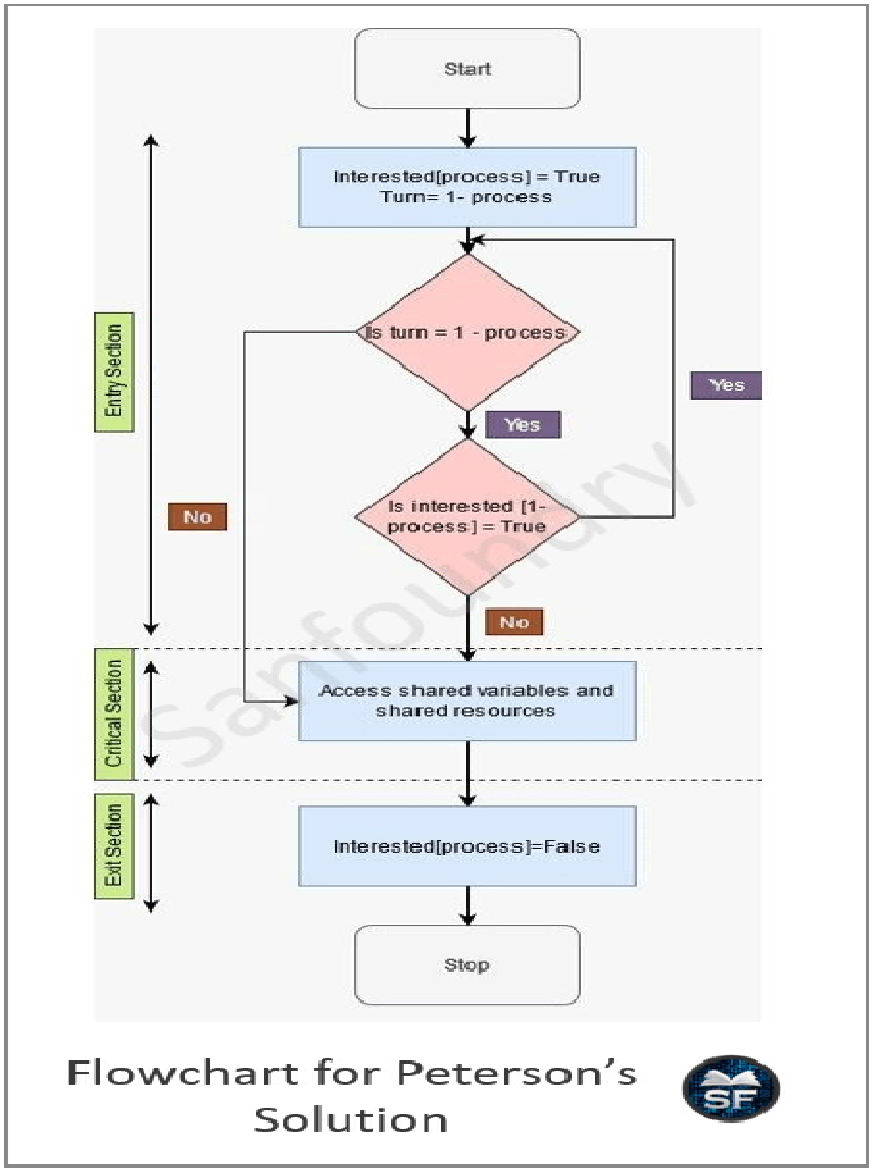
Properties Achieved by Peterson's Solution

1. **Mutual Exclusion**
   Only one process can be inside the critical section at any time.

2. **Progress**
   If no process is inside the critical section, one of the waiting processes is allowed to enter.
3. **Bounded Waiting**
   Each process will get a fair chance to enter the critical section without indefinite postponement.

Limitations of Peterson's Solution

- Applicable only to two processes
- Uses busy waiting, which wastes CPU time
- Not suitable for modern multiprocessor systems



The given flowchart represents the **working of Peterson's Solution** used to achieve mutual exclusion between two processes. It clearly shows how a process enters and exits the critical section safely.

At the **Start**, the process begins execution and decides to enter the critical section.
In the **Entry Section**, the process sets `interested[process] = true`, indicating that it wants to enter the critical section. Then it assigns `turn = 1 - process`, which gives priority to the other process.

After this, the process checks the condition **"Is turn = 1 − process?"**.
If **Yes**, it means the other process has priority, so the current process checks another condition: **"Is interested[1 − process] = true?"**

- If this is **Yes**, the other process is also interested, so the current process keeps waiting (busy waiting).
- If this is **No**, it means the other process is not interested, and the current process can proceed.

If the condition **"Is turn = 1 − process?"** is **No**, the current process is allowed to proceed directly.

Once all conditions are satisfied, the process enters the **Critical Section**, where it accesses shared variables and shared resources. At this point, mutual exclusion is guaranteed because the other process is prevented from entering the critical section.

After completing the critical section, the process moves to the **Exit Section**, where it sets `interested[process] = false`. This indicates that the process no longer wants to enter the critical section and allows the other process to proceed.

Finally, the process reaches the **Stop** state.

This flowchart ensures that:

- Only one process enters the critical section at a time (mutual exclusion)
- Both processes get a fair chance (bounded waiting)
- No deadlock occurs as at least one process can always proceed

Q.3 (a) Differentiate Pre-emptive and Non-Preemptive process scheduling.

Parameter	Pre-emptive Scheduling	Non-Pre-emptive Scheduling
Basic Meaning	In pre-emptive scheduling, the operating system can interrupt a running process and allocate the CPU to another process.	In non-pre-emptive scheduling, once a process gets the CPU, it runs until it completes or enters waiting state.
CPU Control	CPU control can be taken away from a process at any time.	CPU control cannot be taken away until the process finishes execution.
Response Time	Provides faster response time.	Response time is comparatively slow.
Flexibility	More flexible and suitable for interactive systems.	Less flexible and suitable for batch systems.

Parameter	Pre-emptive Scheduling	Non-Pre-emptive Scheduling
CPU Utilization	Better CPU utilization.	Lower CPU utilization compared to pre-emptive scheduling.
Complexity	More complex to implement.	Simple to implement.
Starvation	Chances of starvation are higher.	Chances of starvation are very low.
Examples	Round Robin, Shortest Remaining Time First.	First Come First Serve, Non-pre-emptive SJF.

(b) Explain the difference between internal and external fragmentation.

Parameter	Internal Fragmentation	External Fragmentation
Definition	Internal fragmentation is the wastage of memory that occurs inside an allocated memory block.	External fragmentation is the wastage of memory that occurs between allocated memory blocks.
Cause	Occurs when allocated memory block is larger than the memory required by a process.	Occurs when free memory is divided into small non-contiguous blocks.
Memory Wastage Location	Inside the allocated block.	Outside the allocated blocks.
Occurs In	Fixed partitioning and paging systems.	Dynamic partitioning systems.
Contiguous Memory Issue	Does not require contiguous memory to cause wastage.	Occurs due to lack of a large contiguous memory block.
Total Free Memory	Total free memory may not be usable due to internal wastage.	Total free memory may be sufficient but unusable due to fragmentation.
Solution	Use variable block size or better memory allocation techniques.	Compaction is used to reduce external fragmentation.
Example	Process needs 18 KB but gets 20 KB, 2 KB is wasted.	Free blocks of 5 KB, 10 KB, and 8 KB cannot satisfy a 20 KB request.

(c) What is semaphore? Solve the producer consumer problem with Semaphore.

Semaphore:

A **semaphore** is a **synchronization tool** used in operating systems and concurrent programming to **control access to shared resources** by multiple processes or threads. In a multitasking environment, when several processes attempt to access the same resource (like memory, files, or buffers) simultaneously, it can lead to **race conditions**, data inconsistency, or system crashes. Semaphores are used to **prevent these problems** and ensure **safe and coordinated execution** of processes.

A semaphore is essentially a **non-negative integer variable** that is accessed through two atomic operations:

1. **wait()** (also called **P()** or **down()**) – decreases the semaphore value. If the value becomes negative, the process that invoked the wait operation is **blocked** until the value becomes positive.
2. **signal()** (also called **V()** or **up()**) – increases the semaphore value. If any processes are blocked waiting on the semaphore, one of them is **woken up** to proceed.

Key Features of Semaphore:

- Provides a **mechanism to avoid race conditions** in critical sections.
- Ensures **process synchronization**, so processes execute in the correct order.
- Can be used to solve classical synchronization problems like **Producer-Consumer**, **Reader-Writer**, and **Dining Philosophers** problems.
- Does **not store information about which process is waiting**; it only maintains a counter and a queue of blocked processes.

Producer-Consumer Problem

The **Producer-Consumer Problem** is a classic synchronization problem where:

- **Producer**: Produces items and puts them into a shared buffer.
- **Consumer**: Consumes items from the shared buffer.
- **Constraint**:
  - Producer must wait if buffer is full.
  - Consumer must wait if buffer is empty.

We can solve this problem using **Semaphores**.

Solution Using Semaphores

Semaphores Used:

1. **empty** → Counts the number of empty slots in the buffer (initially = buffer size).
2. **full** → Counts the number of filled slots in the buffer (initially = 0).
3. **mutex** → Binary semaphore to ensure mutual exclusion while accessing the buffer.

Pseudo Code

```
Semaphore empty = n; // n = size of buffer
Semaphore full = 0;
Semaphore mutex = 1; // binary semaphore

Producer() {
    while (true) {
        item = produce_item();      // Produce an item
        wait(empty);                // Decrease empty count
        wait(mutex);                // Enter critical section

        insert_item(item);          // Add item to buffer
```

```
        signal(mutex);              // Exit critical section
        signal(full);               // Increase full count
    }
}

Consumer() {
    while (true) {
        wait(full);                 // Decrease full count
        wait(mutex);                // Enter critical section

        item = remove_item();       // Remove item from buffer

        signal(mutex);              // Exit critical section
        signal(empty);              // Increase empty count

        consume_item(item);         // Consume the item
    }
}
```

Explanation of Flow

1. **Producer** checks for an empty slot using `wait(empty)`.
2. **Producer** enters critical section using `wait(mutex)`.
3. **Producer** adds item to buffer.
4. **Producer** leaves critical section using `signal(mutex)` and signals `full` to indicate a new item is available.
5. **Consumer** waits for `full` to be greater than 0.
6. **Consumer** enters critical section using `wait(mutex)` and removes the item.
7. **Consumer** leaves critical section using `signal(mutex)` and signals `empty` to indicate a free slot.

This ensures:

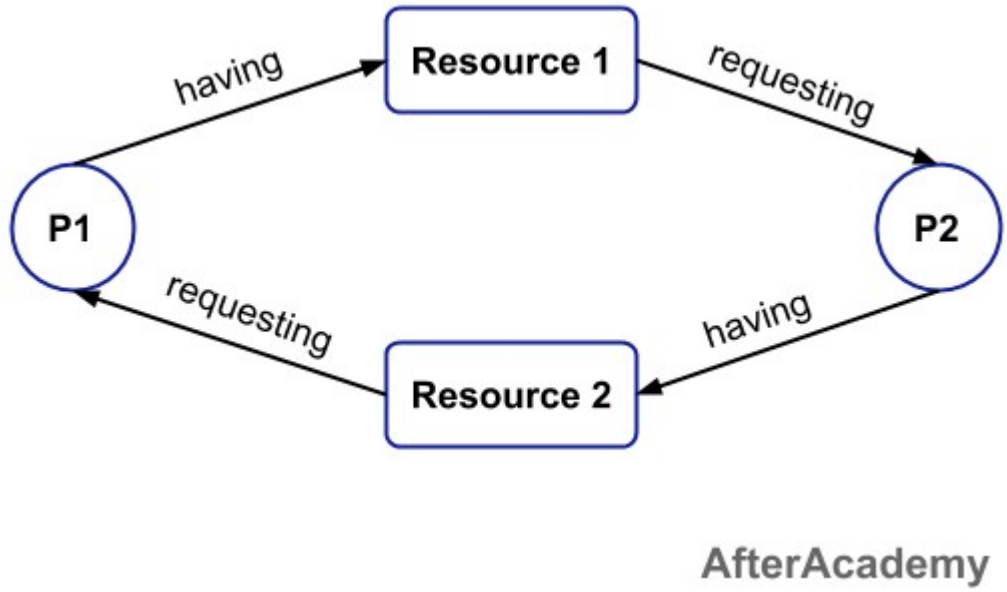
- No race condition (mutex).
- Producer waits if buffer is full (empty).
- Consumer waits if buffer is empty (full).

OR

Q.3 (a) What is deadlock? List the conditions that lead to deadlock.

Deadlock

A **deadlock** is a situation in an operating system where **two or more processes are unable to proceed** because each is **waiting for a resource that the other holds**. In other words, processes are **permanently blocked**, and none of them can continue execution. Deadlocks commonly occur in **multiprogramming systems** where processes compete for **limited resources** like CPU, memory, printers, or files.



Necessary Conditions for Deadlock

For a deadlock to occur, all **four of the following conditions** must hold simultaneously:

1. **Mutual Exclusion:**

   At least one resource must be **non-sharable**, i.e., only one process can use it at a time.

   If another process requests it, it must wait.

2. **Hold and Wait:**

   A process is holding at least **one resource** and is **waiting to acquire additional resources** that are currently held by other processes.

3. **No Preemption:**

   Resources cannot be forcibly taken away from a process. A resource can only be released voluntarily by the process holding it.

4. **Circular Wait:**

   A set of processes exist such that **P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3**, and so on, until the last process **Pn is waiting for a resource held by P1**, forming a circular chain of waiting.

All four conditions must be present for a deadlock to occur. If any one of these conditions is broken, deadlock cannot happen.

(b) Explain Monitor construct to achieve mutual exclusion.

Monitor Construct to Achieve Mutual Exclusion

A **Monitor** is a high-level synchronization construct provided by operating systems or programming languages to **control access to shared resources** and **achieve mutual exclusion** automatically. It is designed to simplify the process of writing concurrent programs, avoiding explicit use of semaphores or locks.

Definition

A **Monitor** is an **abstract data type** that:

- Encapsulates **shared variables**.
- Encapsulates **procedures/functions** that operate on these variables.
- Ensures that **only one process can execute any of the monitor's procedures at a time**, automatically providing **mutual exclusion**.

In other words, the monitor **acts like a controlled environment** where processes can safely access shared resources without causing race conditions.

Key Features of Monitor

1. **Mutual Exclusion:**
   - Only one process can execute inside the monitor at a time.
   - This is automatically enforced; programmers do not need to explicitly acquire or release locks.
2. **Condition Variables:**
   - Monitors provide **condition variables** to allow processes to **wait and signal**.
   - Two main operations on condition variables:
     - `wait(condition)` → Process releases the monitor and waits until it is signaled.
     - `signal(condition)` → Wakes up a waiting process on the condition variable.
3. **Encapsulation:**
   - Shared data is **private** to the monitor.
   - Access is only through **monitor procedures**, which prevents direct interference from outside processes.

Using Monitor for Mutual Exclusion

To achieve **mutual exclusion**:

1. Define a **monitor** for the shared resource.
2. Declare **shared variables** inside the monitor.
3. Provide **procedures** to manipulate these variables.
4. Use **condition variables** if processes need to **wait** for a certain condition (e.g., buffer not full, buffer not empty).
5. Any process calling a monitor procedure **automatically enters a critical section**, and no other process can execute inside the monitor until the first one exits.

Example: Bounded Buffer Using Monitor

```
monitor BoundedBuffer {
    int buffer[N];      // Shared buffer
    int count = 0;      // Number of items
    condition full, empty;

    procedure insert(item) {
        if (count == N)
            wait(empty);      // Wait if buffer full
        buffer[count] = item;
        count = count + 1;
        signal(full);         // Signal that buffer has item
    }

    procedure remove() {
        if (count == 0)
            wait(full);       // Wait if buffer empty
        item = buffer[count-1];
        count = count - 1;
        signal(empty);        // Signal that buffer has space
        return item;
    }
}
```

Explanation:

- **Mutual exclusion** is automatically provided by the monitor.
- **Condition variables** `full` and `empty` handle the waiting and signaling when the buffer is full or empty.
- Producers call `insert(item)` and consumers call `remove()` safely without explicit semaphores.

Key Advantages:

- Simplifies **concurrent programming**.

- Avoids **explicit lock management**.
- Reduces the chances of **race conditions and deadlocks** (if used properly).

(c) Solve the Dining philosopher problem with semaphore.

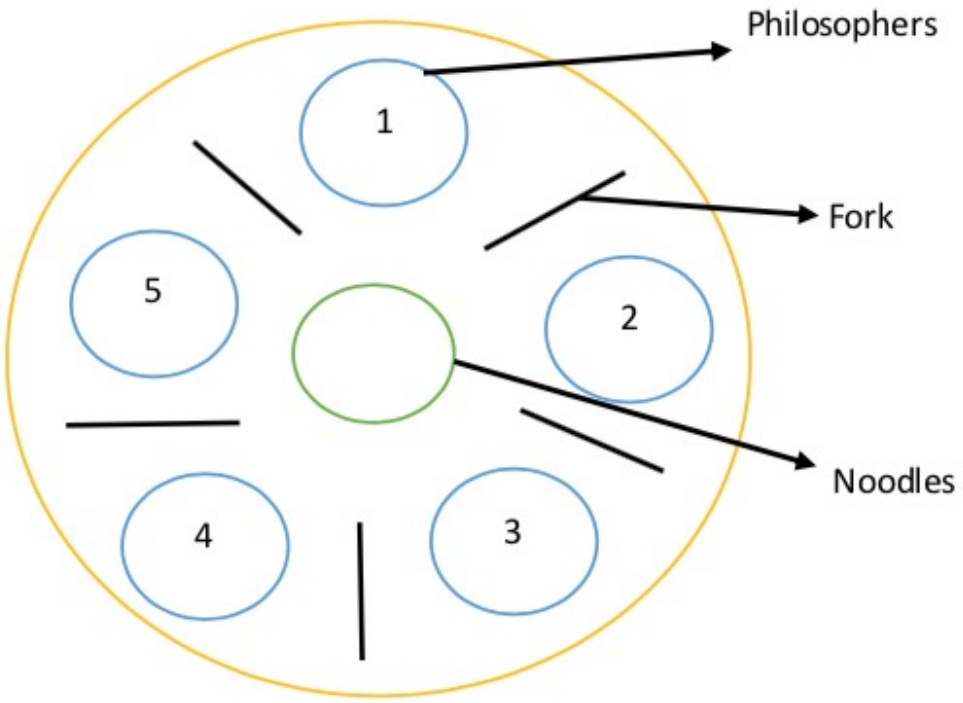


Fig: Dining Philosopher's Problem

Dining Philosophers Problem

The **Dining Philosophers Problem** is a classical synchronization problem proposed by **Edsger Dijkstra** to illustrate issues of **deadlock, starvation, and resource sharing**.

Problem Statement:

- Imagine **five philosophers** sitting around a circular table.
- Each philosopher **alternates between thinking and eating**.
- Between every pair of philosophers, there is **one fork** (or chopstick).
- To **eat**, a philosopher needs **both the left and right forks**.
- Only one philosopher can hold a fork at a time.

Goal:
Design a solution where **no two philosophers use the same fork simultaneously**, **no deadlock occurs**, and **no philosopher starves**.

Solution Using Semaphores

We use **semaphores** to control access to forks.

Semaphores Used:

1. **Fork Semaphores:**
   - `Semaphore fork[5]` → Represents each fork.
   - Initialized to `1` (available).
2. **Mutex Semaphore (optional):**
   - Ensures that picking up forks is done **atomically** to prevent deadlock in some implementations.

Steps / Algorithm (Using Semaphore):

1. Each philosopher tries to pick up the **left fork**.
2. `wait(fork[i]);  // i = philosopher number`
3. Then, the philosopher tries to pick up the **right fork**.
4. `wait(fork[(i+1) % 5]);`
5. The philosopher **eats**.
6. After eating, the philosopher **puts down the forks**:
7. `signal(fork[i]);`
8. `signal(fork[(i+1) % 5]);`
9. Then, the philosopher **thinks** before repeating the cycle.

C Code Example Using Semaphore

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5  // Number of philosophers

sem_t fork[N];

void* philosopher(void* num) {
    int i = *(int*)num;
    while(1) {
        printf("Philosopher %d is thinking\n", i);
        sleep(1); // Thinking

        // Pick up forks
        sem_wait(&fork[i]);                  // Left fork
        sem_wait(&fork[(i+1) % N]);          // Right fork

        printf("Philosopher %d is eating\n", i);
        sleep(2); // Eating

        // Put down forks
```

```
        sem_post(&fork[i]);
        sem_post(&fork[(i+1) % N]);

        printf("Philosopher %d finished eating and resumes thinking\n", i);
    }
}

int main() {
    pthread_t phil[N];
    int i;
    int id[N];

    // Initialize semaphores
    for(i = 0; i < N; i++)
        sem_init(&fork[i], 0, 1);

    // Create philosopher threads
    for(i = 0; i < N; i++) {
        id[i] = i;
        pthread_create(&phil[i], NULL, philosopher, &id[i]);
    }

    // Join threads
    for(i = 0; i < N; i++)
        pthread_join(phil[i], NULL);

    return 0;
}
```

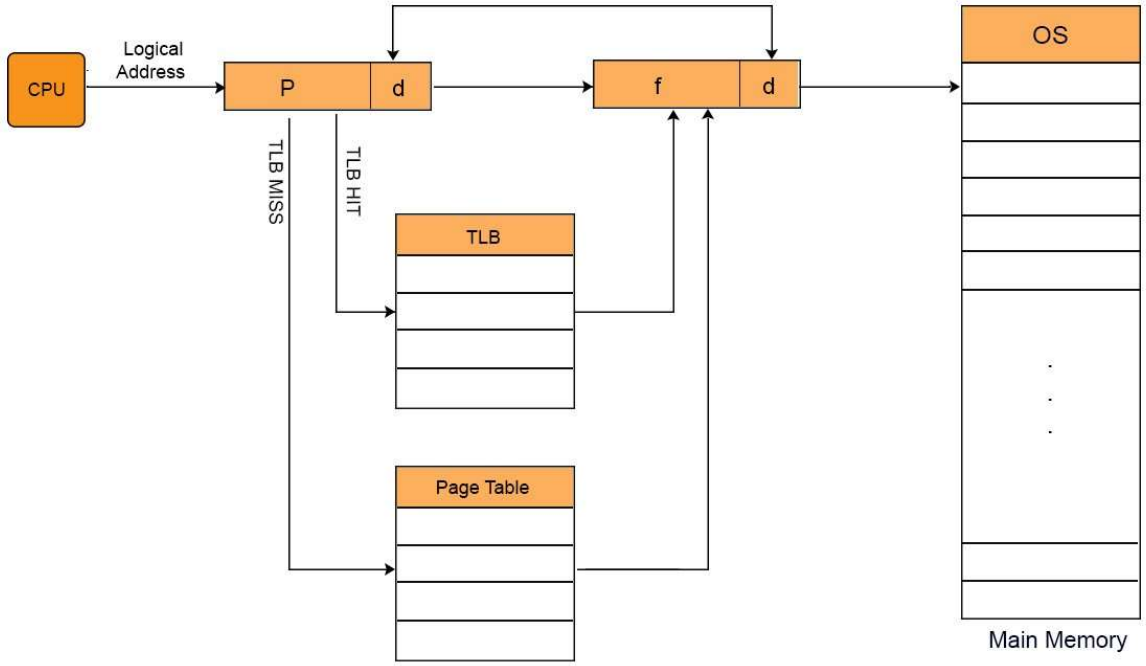
Explanation of Solution

1. **Mutual Exclusion:**
   - Each fork is a semaphore, ensuring **only one philosopher can pick it at a time**.
2. **Avoiding Deadlock:**
   - If each philosopher picks up the left fork first, deadlock can occur.
   - **Solution:** Make one philosopher pick up **right fork first** or use a **mutex** to ensure atomic picking.
3. **Synchronization:**
   - Semaphores block philosophers if a fork is not available.
   - Philosophers automatically wait without busy-waiting.
4. **Starvation-Free:**
   - Using proper semaphore order ensures every philosopher **eventually gets to eat**.

Q.4 (a) Explain the difference between logical and physical addresses.

Feature	Logical Address	Physical Address
Also Called	Virtual Address – the address generated by the CPU while a program is running.	Real Address / Actual Address – the address where the data actually resides in main memory (RAM).
Generated By	Generated by the CPU during program execution when instructions refer to memory locations.	Generated after translation by the Memory Management Unit (MMU); represents actual location in RAM.
Visibility	Visible to the user or programmer; used in the code to refer to variables and instructions.	Not visible to the user/program; handled internally by the operating system and hardware.
Mapping	Needs to be mapped to a physical address by the MMU before accessing main memory.	Already represents the exact memory location; no further mapping is needed.
Purpose / Use	Used by the program to access instructions and data in a **logical or virtual sense**.	Used by hardware to fetch or store data from/to the **actual physical memory**.
Access Control / Protection	Logical addresses allow OS to provide process isolation and memory protection.	Physical addresses are directly used by the memory hardware; cannot enforce isolation by themselves.
Example	A program may reference address 1000 in its code (logical).	That logical address 1000 may correspond to physical memory location 5000 in RAM.
Memory Management	Enables features like **virtual memory, paging, and process isolation**.	Represents actual RAM location where instructions and data reside; no abstraction layer.
Dependence	Independent of the actual hardware memory configuration.	Dependent on the actual RAM installed and its organization.
Translation Requirement	Must be translated to physical address before memory access.	No translation needed; directly used by memory circuitry.

(b) What is TLB(Translation Lookaside Buffer), and what is use of it?



Translation Lookaside Buffer (TLB)

A **Translation Lookaside Buffer (TLB)** is a **high-speed, small memory cache** used by the **Memory Management Unit (MMU)** in modern computer systems to speed up the translation of **logical (virtual) addresses to physical addresses**. It is an essential component in systems that use **paging and virtual memory**.

In a virtual memory system, every memory access requires the CPU to generate a **logical address**, which consists of two parts:

1. **Page Number (P):** Identifies which page of the process's virtual memory is being accessed.
2. **Page Offset (d):** Identifies the specific location within that page.

Normally, the CPU must look up the **page table** to map the page number `P` to the corresponding **frame number (f)** in physical memory. However, accessing the page table in

main memory for every instruction is **time-consuming**, because it requires at least **two memory accesses**:

- One to access the page table
- One to access the actual data in main memory

How TLB Works

The **TLB** acts as a **cache for page table entries**:

1. **CPU generates a logical address:** `P | d`
   - `P` = page number
   - `d` = page offset
2. **MMU checks TLB for the page number (P):**
   - **TLB Hit:** If the page number exists in the TLB, the corresponding **frame number (f)** is retrieved immediately.
     - The physical address is then formed as `f | d` and used to access **main memory**.
     - This avoids a full page table lookup, **reducing memory access time** significantly.
   - **TLB Miss:** If the page number is not in the TLB:
     - The MMU consults the **page table** in main memory to find the corresponding frame number `f`.
     - The TLB is updated with this new page → frame mapping for future accesses.
     - The physical address `f | d` is then used to access memory.
3. **Memory Access:**
   - Once the physical address is obtained, the CPU can **read or write data** in main memory.

This process is illustrated in your diagram, showing **logical address from CPU → TLB check → page table (if miss) → physical address → main memory**.

Purpose and Use of TLB

1. **Speeds up address translation:**
   - Most memory accesses refer to recently used pages, so storing their translations in the TLB reduces the time needed to obtain the physical address.
2. **Reduces CPU wait time:**
   - Without a TLB, each memory access may require **two or more memory accesses**, slowing down execution.
   - With TLB hits, only **one memory access** is needed, improving overall performance.
3. **Supports virtual memory efficiently:**

- Makes virtual memory practical by allowing fast translation of logical addresses, even with large page tables.

4. **Minimizes page table lookups:**
   - The TLB acts as a **fast lookup cache**, so the page table in main memory is accessed only on TLB misses.
5. **Reduces system overhead:**
   - Reduces the number of memory operations, which is critical in high-performance systems with large address spaces.

(c) Explain following process scheduling algorithms. i) First Come First Serve ii) Non-preemptive Shortest Job First iii) Shortest Remaining Time Next iv) Round Robin

i) First Come First Serve

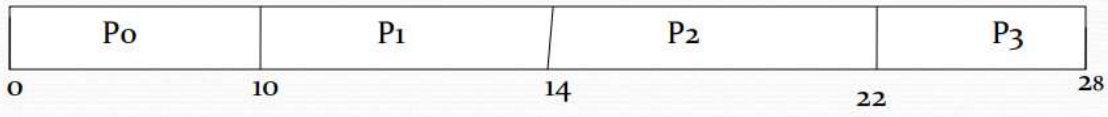
First-Come, First-Served Scheduling Algorithm is nonpreemptive algorithm.

- It is the simplest of all the scheduling algorithms.
- The key concept of this algorithm is "The process which comes first in the ready queue will allocate the CPU first".
- The next process will allocate the CPU only after the previous gets fully executed.

Sr. No	Process	Execution Time/ Burst Time(ms)
1	p0	10
2	p1	4
3	p2	8
4	p3	6

If the arrival time is not given. We can assume arrival time of all process is 0.

Gantt Chart:



Completion Time: P0 = 10, P1 =14, P2 = 22, P3 =28

Turnaround Time(TAT) = Completion Time(CT) – Arrival Time(AT)

Turnaround Time(TAT) of each process

P0 : 10 – 0 = 10 ms

P1 : 14 – 0 = 14 ms

P2 : 22 – 0 = 22 ms

P3 : 28 – 0 = 28 ms

Average Turnaround Time(ATAT) = Total turn around time of all processes / Total no of processes
Average Turnaround Time(ATAT) = (10+14+22+28) / 4 = 18.5 ms

Waiting Time(WT) = Turnaround Time(TAT) – Burst Time(BT)

Waiting Time(WT) of each process

P0 : 10 – 10 = 0 ms

P1 : 14 – 4 = 10 ms

P2 : 22– 8 = 14 ms

P3 : 28 – 6 = 22 ms

Average Waiting Time(AWT) = Total waiting time of all processes / Total no of processes

Average Waiting Time(AWT) = (0+10+14+22) / 4 = 11.5 ms

ii) Non-preemptive Shortest Job First:

Completion Time:

P0 =28, P1 = 4, P2 =18 P3 = 10

Turnaround Time(TAT) = Completion Time(CT) – Arrival Time(AT)

Turnaround Time(TAT) of each process

P0 : 28 – 0 = 28 ms

P1 : 4 – 0 = 4 ms

P2 : 18 – 0 = 18 ms

P3 : 10 – 0 = 10 ms

Average Turnaround Time(ATAT) = Total turn around time of all processes / Total no of processes
Average Turnaround Time(ATAT) = (28+4+18+10) / 4 = 15 ms

Waiting Time(WT) = Turnaround Time(TAT) – Burst Time(BT)

Waiting Time(WT) of each process

P0 : 28-10 = 18 ms

P1 : 4 – 4 = 0 ms

P2 : 18– 8 = 10 ms

P3 : 10 – 6 = 4 ms

Average Waiting Time(AWT) = Total waiting time of all processes / Total no of processes

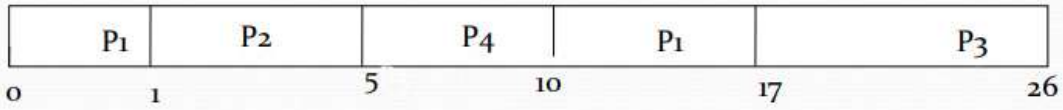
Average Waiting Time(AWT) = (18+0+10+4) / 4 = 8 ms

iii) Shortest Remaining Time Next

Preemptive SJF Algorithm

Sr. No	Process	Arrival Time	Execution Time/ Burst Time(ms)
1	po	0	8
2	p1	1	4
3	p2	2	9
4	p3	3	5

Gantt Chart



Completion Time:

P0 =17, P1 = 5, P2 =26 P3 = 10

Turnaround Time(TAT) = Completion Time(CT) – Arrival Time(AT)

Turnaround Time(TAT) of each process

P0 : 17 – 0 = 17 ms
P1 : 5 – 1 = 4 ms
P2 : 26 – 2 = 24 ms
P3 : 10 – 3 = 7 ms

Average Turnaround Time(ATAT) = Total turn around time of all processes / Total no of processes

Average Turnaround Time(ATAT) = (17+4+24+7) / 4 = 13 ms

Waiting Time(WT) = Turnaround Time(TAT) – Burst Time(BT)

Waiting Time(WT) of each process

P0 : 17-8 = 9 ms
P1 : 4 – 4 = 0 ms
P2 : 24 -9 = 15 ms
P3 : 7 – 5 = 2 ms

Average Waiting Time(AWT) = Total waiting time of all processes / Total no of processes

Average Waiting Time(AWT) = (9+0+15+2) / 4 = 6.5 ms

iv) Round Robin :

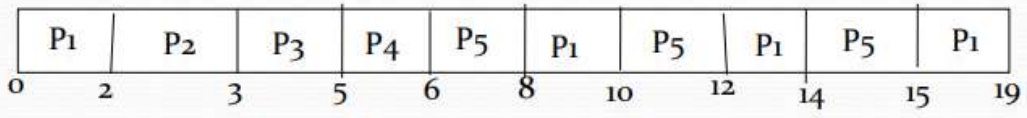
It is designed especially for time sharing systems. Here CPU switches between the processes. When the time quantum expired, the CPU switched to another job. A small unit of time, called a time quantum or time slice. A time quantum is generally from 10 to 100 ms. The time quantum is generally depending on OS. Here ready queue is a circular queue. CPU scheduler picks the first process from ready queue, sets timer to interrupt after one time quantum and dispatches the process.

Round-Robin Scheduling Algorithm

Sr. No	Process	Burst time
1	P_1	10
2	P_2	1
3	P_3	2
4	P_4	1
5	P_5	5

Time Quantum is 2 ms.

Gantt Chart:



Round-Robin Scheduling Algorithm

Completion Time:

P1 = 19, P2 =3, P3 = 5, P4= 6, P5 = 15

Turnaround Time(TAT) = Completion Time(CT) – Arrival Time(AT)

Turnaround Time(TAT) of each process

P1 : 19 – 0 = 19 ms
P2 : 3 – 0 = 3 ms
P3 : 5 – 0 = 5 ms
P4: 6 – 0 = 6 ms
P5 : 15 – 0 = 15 ms

Average Turnaround Time(ATAT) = Total turn around time of all processes / Total no of processes

Average Turnaround Time(ATAT) = (19+3+5+6+15) / 5 = 9.6 ms

Waiting Time(WT) = Turnaround Time(TAT) – Burst Time(BT)

Waiting Time(WT) of each process

P1 : 19 – 10 = 9 ms
P2 : 3 – 1 = 2 ms
P3 : 5– 2 = 3 ms
P4: 6– 1 = 5 ms
P5 : 15– 5 = 10 ms

Average Waiting Time(AWT) = Total waiting time of all processes / Total no of processes

Average Waiting Time(AWT) = (9+2+3+5+10) / 5 = 5.8 ms

OR

Q.4 (a) What is virtualization? Explain the benefits of virtualization.

Virtualization is a technology that allows a **single physical computer (hardware system)** to be divided into **multiple virtual machines (VMs)**. Each virtual machine behaves like an independent computer with its own **operating system, applications, and resources**, even though all VMs share the same physical hardware.

Virtualization is achieved using a software layer called a **hypervisor** (also known as Virtual Machine Monitor – VMM). The hypervisor sits between the hardware and the operating systems and manages the allocation of CPU, memory, storage, and other resources to each virtual machine.

In simple words, virtualization enables **multiple operating systems to run simultaneously on one physical machine**, making better use of hardware resources.

Benefits of Virtualization

1. Better Hardware Utilization

- In traditional systems, one OS uses only a small portion of hardware capacity.
- Virtualization allows multiple VMs to run on the same hardware.
- This ensures maximum use of CPU, memory, and storage.

---

2. Cost Reduction

- Reduces the number of physical servers required.
- Saves money on hardware purchase, maintenance, power consumption, and cooling.
- Lower infrastructure and operational costs.

---

3. Server Consolidation

- Multiple underutilized physical servers can be combined into a single physical server with multiple virtual machines.
- Reduces data center space and hardware clutter.

---

4. Isolation and Security

- Each virtual machine is isolated from others.
- Failure or crash in one VM does not affect other VMs.
- Improves system stability and security.

---

5. Easy Backup and Recovery

- Virtual machines can be backed up as files.
- In case of system failure, VMs can be restored quickly.
- Supports disaster recovery and business continuity.

---

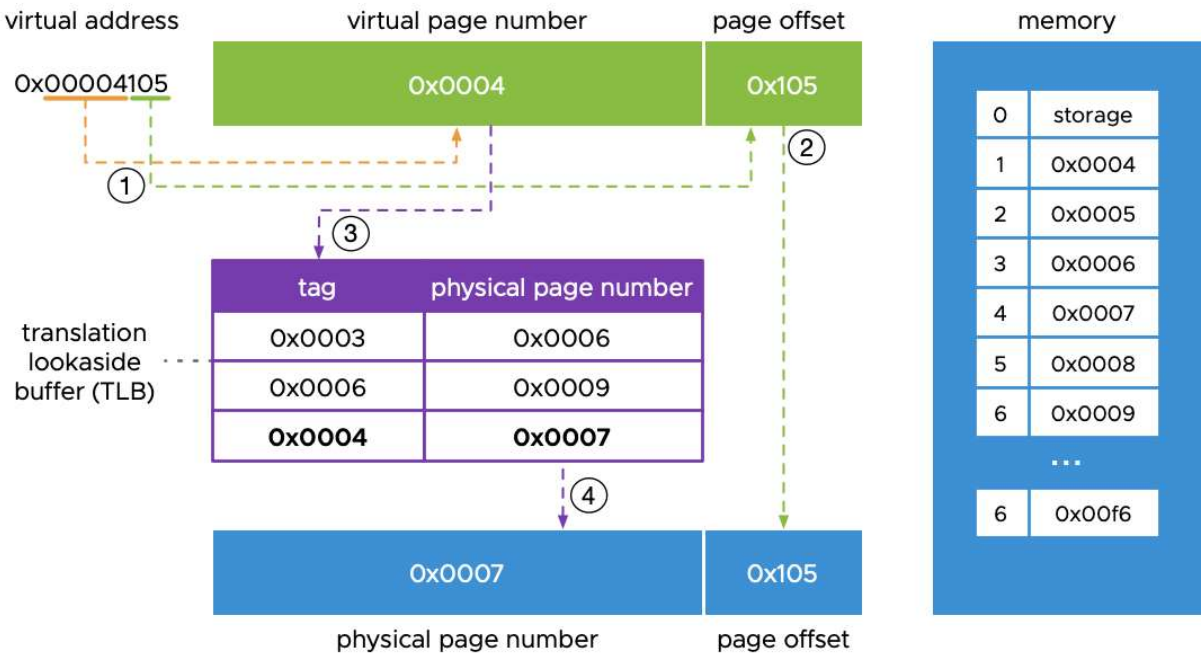
6. Faster Deployment

- New servers or environments can be set up quickly using virtual machine templates.
- Reduces deployment time compared to physical system

(b) Define Virtual Memory. Explain the process of converting virtual addresses to physical addresses with a neat diagram.

Virtual Memory is a memory management technique that allows a computer system to execute programs that are **larger than the size of physical main memory (RAM)**. It creates an illusion for the user that the system has **large continuous memory**, while actually using a combination of **main memory (RAM)** and **secondary storage (hard disk/SSD)**.

In virtual memory, a program is divided into fixed-size blocks called **pages**, and physical memory is divided into **frames**. Only the required pages of a program are loaded into main memory, while the remaining pages stay on secondary storage.



Conversion of Virtual Address to Physical Address

The conversion of a virtual address into a physical address is handled by the **Memory Management Unit (MMU)** with the help of a **page table**.

Step-by-Step Address Translation Process

1. **Virtual Address Generation**
    - The CPU generates a **virtual address**.
    - The virtual address consists of:
        - **Page Number (p)**
        - **Offset (d)**
2. **Page Table Lookup**
    - The page number is used as an index into the **page table**.
    - The page table contains the **frame number** corresponding to each page.
3. **Valid/Invalid Check**
    - If the page table entry is **valid**, the page is present in main memory.
    - If it is **invalid**, a **page fault** occurs.
4. **Page Fault Handling (if required)**
    - The required page is fetched from secondary storage into a free frame.
    - The page table is updated.
    - The instruction is restarted.
5. **Physical Address Formation**
    - The frame number obtained from the page table is combined with the offset.
    - This forms the **physical address**.

Address Format

- **Virtual Address** = Page Number | Offset
- **Physical Address** = Frame Number | Offset

Advantages of Virtual Memory

- Efficient use of physical memory.
- Programs are not limited by RAM size.
- Simplifies memory management.
- Reduces swapping overhead.

(c) Consider a Swapping system in which memory consists of the following hole sizes(in MB) in memory order: 10, 4,20,18,7,9,12,15. Which hole is taken for successive segment request of 12 MB and 10 MB for First fit, Best fit, Worst fit and Next fit.

Given:

Memory holes (in MB) in **memory order**:
10, 4, 20, 18, 7, 9, 12, 15

Segment requests (in order):

1. **12 MB**

2. **10 MB**

---

1) First Fit

Rule:
Allocate the first hole (from the beginning of memory) that is large enough.

Allocation for 12 MB

- 10 ✗
- 4 ✗
- **20 ✓ → selected**

Remaining hole after allocation:
20 − 12 = **8 MB**

Updated holes:

10, 4, **8**, 18, 7, 9, 12, 15

Allocation for 10 MB

- **10 ✓ → selected**

First Fit Result:

- 12 MB → **20 MB hole**
- 10 MB → **10 MB hole**

---

2) Best Fit

Rule:
Allocate the **smallest hole** that is large enough.

Allocation for 12 MB

Eligible holes:
20, 18, **12**, 15

- **12 MB hole is exact fit → selected**

Remaining hole:
12 − 12 = **0 MB (removed)**

Updated holes:

10, 4, 20, 18, 7, 9, 15

Allocation for 10 MB

Eligible holes:
10, 20, 18, 15

- **10 MB hole is exact fit → selected**

Best Fit Result:

- 12 MB → **12 MB hole**
- 10 MB → **10 MB hole**

---

3) Worst Fit

Rule:
Allocate the **largest available hole**.

Allocation for 12 MB

Largest hole = **20 MB → selected**

Remaining hole:
20 − 12 = **8 MB**

Updated holes:

10, 4, **8**, 18, 7, 9, 12, 15

Allocation for 10 MB

Largest hole = **18 MB → selected**

Remaining hole:
18 − 10 = **8 MB**

Worst Fit Result:

- 12 MB → **20 MB hole**
- 10 MB → **18 MB hole**

---

4) Next Fit

Rule:
Similar to First Fit, but **search starts from the last allocated position**, not from the beginning.

Allocation for 12 MB

- Start from beginning
- 10 ✗, 4 ✗
- **20 ✓ → selected**

Remaining hole:
20 − 12 = **8 MB**

☞ Pointer moves to this position.

Updated holes:

10, 4, **8**, 18, 7, 9, 12, 15

Allocation for 10 MB

Start search **after 8 MB hole**:

- 8 ✗
- **18 ✓ → selected**

Remaining hole:
18 − 10 = **8 MB**

Next Fit Result:

- 12 MB → **20 MB hole**
- 10 MB → **18 MB hole**

Q.5 (a) What is Inode? Explain it's usage.

Q.5 (a) What is Inode? Explain its usage

What is Inode?

An **inode (Index Node)** is a **data structure used by Unix and Linux file systems** to store information about a file or directory **except its name and actual data**.

Each file in the file system is associated with a **unique inode number**, which the operating system uses to identify and manage the file.

In simple words, an inode acts as a **record or identity card of a file**, containing all metadata required to access and manage that file.

Information Stored in an Inode

An inode stores the following information:

- File type (regular file, directory, device file, etc.)
- File size
- Owner (User ID)
- Group ID
- File permissions (read, write, execute)
- Number of hard links
- Timestamps:
    - Creation time
    - Last modification time
    - Last access time
- Pointers to data blocks on disk (direct, indirect pointers)

Usage of Inode

1. File Identification

- Each file has a unique inode number.
- The OS uses this number to uniquely identify a file internally, even if the file name changes.

---

2. File Access and Management

- When a file is opened, the system checks the inode to:
    - Verify permissions
    - Locate the disk blocks where file data is stored
- Inodes help the OS read and write file data efficiently.

---

3. Directory Implementation

- Directories store **file names mapped to inode numbers**.
- When a file is searched, the directory gives the inode number, and the inode provides the file's metadata.

---

4. Hard Link Management

- Multiple file names (hard links) can point to the same inode.
- Inode keeps a count of how many links reference it.
- File data is deleted only when link count becomes zero.

---

5. Disk Space Management

- Inodes contain pointers to disk blocks.
- Helps the file system manage disk storage and locate file contents quickly.

---

6. File System Integrity

- Inodes help detect and recover file system errors.
- Tools like `fsck` use inodes to check consistency of the file system.

(b) List Advantages of LINUX/UNIX operating system over Windows.

Advantages of LINUX / UNIX Operating System over Windows

1. **Open Source**

   Linux/UNIX source code is freely available. Users can study, modify, and distribute it, unlike Windows which is proprietary.

2. **Better Security**

   Linux/UNIX has strong user permission and file ownership mechanisms, making it less vulnerable to viruses and malware.

3. **Stability and Reliability**

   Linux/UNIX systems can run for long periods without rebooting. They are widely used in servers due to high stability.

4. **Multiuser Support**

   Multiple users can work on the same system at the same time without interfering with each other.

5. **Multitasking Capability**

   Linux/UNIX efficiently handles multiple processes simultaneously with better process scheduling.

6. **Lower Cost**

   Most Linux distributions are free. There are no licensing fees as compared to Windows.

7. **Efficient Resource Utilization**

   Linux/UNIX performs well even on low-end hardware and uses system resources efficiently.

8. **Powerful Command Line Interface**

   UNIX/Linux provides a strong shell with scripting support, making system administration faster and more flexible.

9. **High Customization**

   Users can customize the desktop environment, kernel, and system behavior according to their needs.

10. **Better Networking Support**

    Built-in networking tools and protocols make Linux/UNIX ideal for server and network applications.

11. **Fewer System Crashes**

    Linux/UNIX systems are less prone to crashes and system hangs compared to Windows.

12. **Wide Developer and Community Support**

    Large global community provides free support, documentation, and frequent updates.

13. **Support for Multiple File Systems**

    Linux/UNIX supports many file systems like ext4, XFS, Btrfs, FAT, NTFS, etc.

14. **Ideal for Servers and Cloud Computing**

    Most web servers, cloud platforms, and supercomputers run on Linux/UNIX due to performance and security.

15. **No Forced Updates**

    Users have full control over system updates, unlike Windows which forces updates.

(c) Explain various Page Replacement Algorithms.

FIFO (First In First Out) Page Replacement

- Simplest page replacement algorithm
- Oldest page in main memory is the one which will be selected for replacement. Easy to implement, keep a list, replace pages from the tail and add new pages

		1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	0
5	5	5	5	2	2	2	2	4	4	4	4	4	4	4	4	7	7	7	7	7
			H				H		H	H		H	H		H			H	H	H

Total Page Fault= 11

Number of Hit= 10

Rate of page fault = No. of page fault/No. of frames

11/3= 3.6

Beledy's Anomaly:

Beledy's anomaly is that the page-fault rate may increase as the number of allocated frames increases.

For example if we consider the following page reference string

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4 with no of frame 3, we get total 9 page fault as shown in fig (a). But if we increase the no. of frame from 3 to 4 we get 10 page fault as shown in fig (b)

Case 1: Frame=3

		2	2	2	1	1	1	1	1	3	3
	1	1	1	0	0	0	0	0	2	2	2
0	0	0	3	3	3	4	4	4	4	4	4
							H	H			H

Fig (a)

Page Fault= 9

Case 2: Frame=4

			3	3	3	3	3	3	2	2	2
		2	2	2	2	2	2	1	1	1	1
	1	1	1	1	1	1	0	0	0	0	4
0	0	0	0	0	0	4	4	4	4	3	3
				H	H						

Fig (b)

Page Fault= 10

Advantages:

- It is simple and easy to understand & implement.

Disadvantages:

- The process effectiveness is low.
- When we increase the number of frames while using FIFO, we are giving more memory to processes. So, page fault should decrease, but here the page faults are increasing. This problem is called as Beledy's anomaly

LRU (Least-Recently-Used) Page Replacement

In this algorithm, the page that has been not used for longest period of time is selected for replacement.

Example: Reference string is

5, 0, 1, 0, 2, 3, 0, 2, 4, 3, 3, 2, 0, 2, 1, 2, 7, 0, 1, 1, 0

		1	1	1	3	3	3	4	4	4	4	0	0	0	0	7	7	7	7	7
	0	0	0	0	0	0	0	0	3	3	3	3	3	1	1	1	0	0	0	0
5	5	5	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1
			H			H	H			H	H		H		H				H	H

Total Page Fault= 12

Number of Hit= 9

Rate of page fault= No. of page fault/No. of frames = 12/3= 4

Advantages:

- It is open for full analysis.
- In this, we replace the page which is least recently used, thus free from Belady's Anomaly.
- Easy to choose page which has faulted and hasn't been used for a long time.

Disadvantages:

- It requires additional Data Structure to be implemented.
- Hardware assistance is high.

Optimal Page Replacement

This algorithm state that: replace that page which will not be used for longest period of time i.e. future knowledge of reference string is required.

Example: Reference string is

5, 0, 1, 0, 2, 3, 0, 2, 4, 3, 3, 2, 0, 2, 1, 2, 7, 0, 1, 1, 0

		1	1	1	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	4	4	4	4	0	0	0	0	0	0	0	0
5	5	5	5	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7	7	7
			H			H	H		H	H	H		H		H		H	H	H	H

Total Page Fault= 9

Number of Hit= 12

Rate of page fault= No. of page fault/No. of frames = 9/3= 3

Advantages:

- Complexity is less and easy to implement.
- Assistance needed is low i.e Data Structure used are easy and light.

Disadvantages:

- Optimal page replacement algorithm is perfect, but not possible in practice as the operating system cannot know future requests.
- Error handling is tough.

OR

Q.5 (a) Explain I/O buffering in brief.

I/O buffering is a technique used by the operating system to **temporarily store data in memory (buffer)** while it is being transferred between an **I/O device** and **main memory or CPU**.
The buffer acts as an intermediate storage area that helps manage the difference in **speed and data transfer rates** between fast CPU/memory and slow I/O devices.

Why I/O Buffering is Needed

- I/O devices are much slower than the CPU.
- Data transfer may be in different sizes or speeds.
- Helps in smooth and efficient data communication.

Types of I/O Buffering

1. **Single Buffering**
   - Uses one buffer between the device and the user process.
   - While one buffer is being filled, the process waits.
   - Simple but less efficient.
2. **Double Buffering**
   - Uses two buffers alternately.
   - While one buffer is being filled, the other is being processed.
   - Reduces waiting time and improves performance.
3. **Circular (Multiple) Buffering**
   - Uses more than two buffers in a circular manner.
   - Suitable for continuous data streams like audio or video.

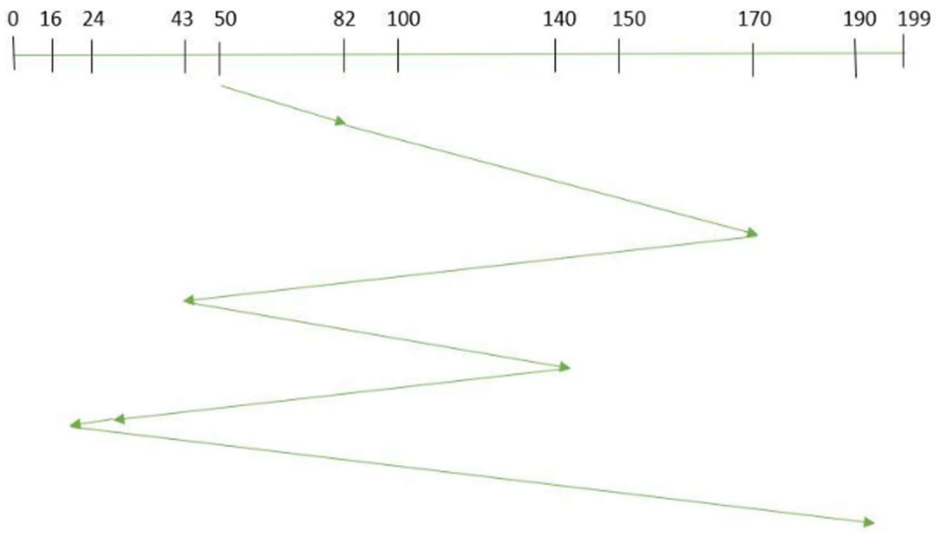
Advantages of I/O Buffering

- Improves system performance.
- Reduces CPU idle time.
- Smoothens data transfer.
- Supports asynchronous I/O operations.

(b) Explain Disk arm scheduling algorithms.

1. FCFS (First Come First Serve)

FCFS is the simplest of all Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.



Suppose the order of request is- (82,170,43,140,24,16,190) and current position of Read/Write head is: 50

So, total overhead movement (total distance covered by the disk arm) = (82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16) =642

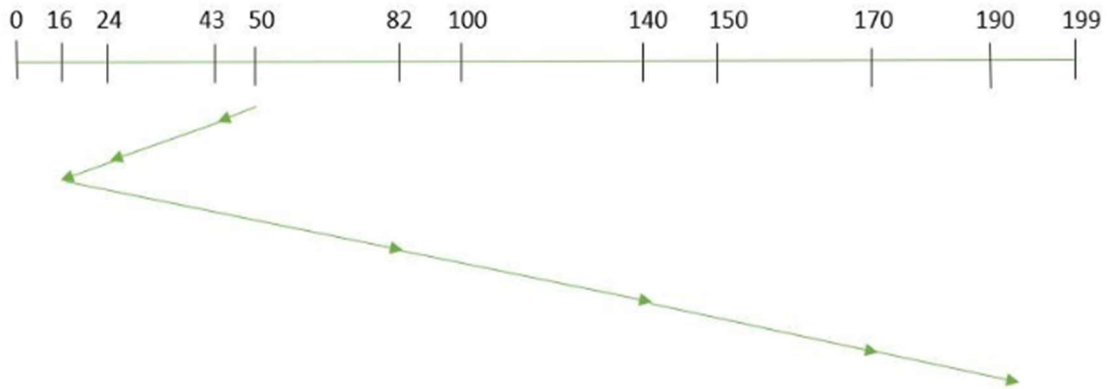
2) Shortest Seek Time First (SSTF)

SSTF selects the disk request that is **closest to the current head position**, i.e., requires minimum seek time.

Working:

- Calculate the distance of each request from the current head position.
- Select the request with the shortest seek distance.

Example:



Shortest Seek Time First

Suppose the order of request is- (82,170,43,140,24,16,190) and current position of Read/Write head is: 50

total overhead movement (total distance covered by the disk arm) = (50-43)+(43-24)+(24-16)+(82-16)+(140-82)+(170-140)+(190-170) =208

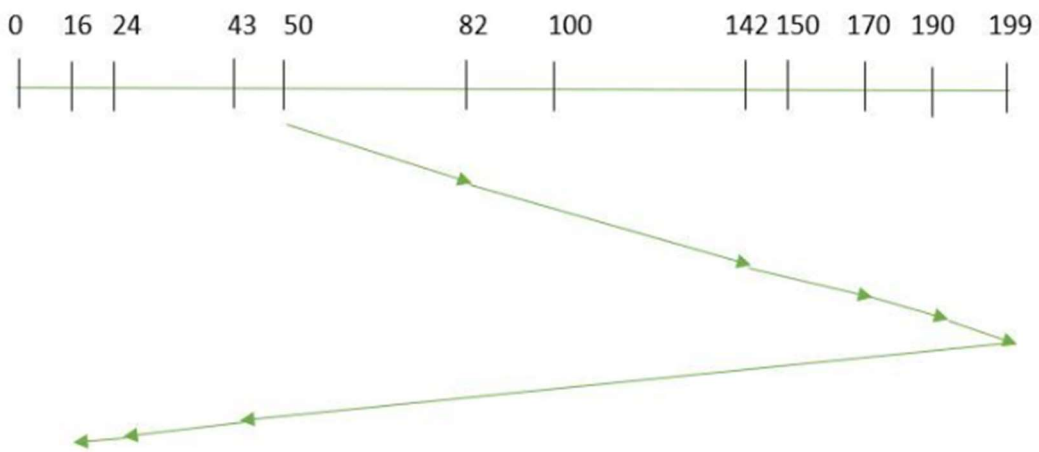
3) SCAN (Elevator Algorithm)

Explanation:
In SCAN scheduling, the disk arm moves in **one direction**, servicing all requests along the way until it reaches the end, then reverses direction.

Working:

- Disk arm moves like an elevator.
- Services all requests in one direction, then changes direction.

Example:



SCAN Algorithm

Suppose the requests to be addressed are-82,170,43,140,24,16,190 and the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**
Therefore, the total overhead movement (total distance covered by the disk arm) is calculated as
= (199-50) + (199-16) = 332

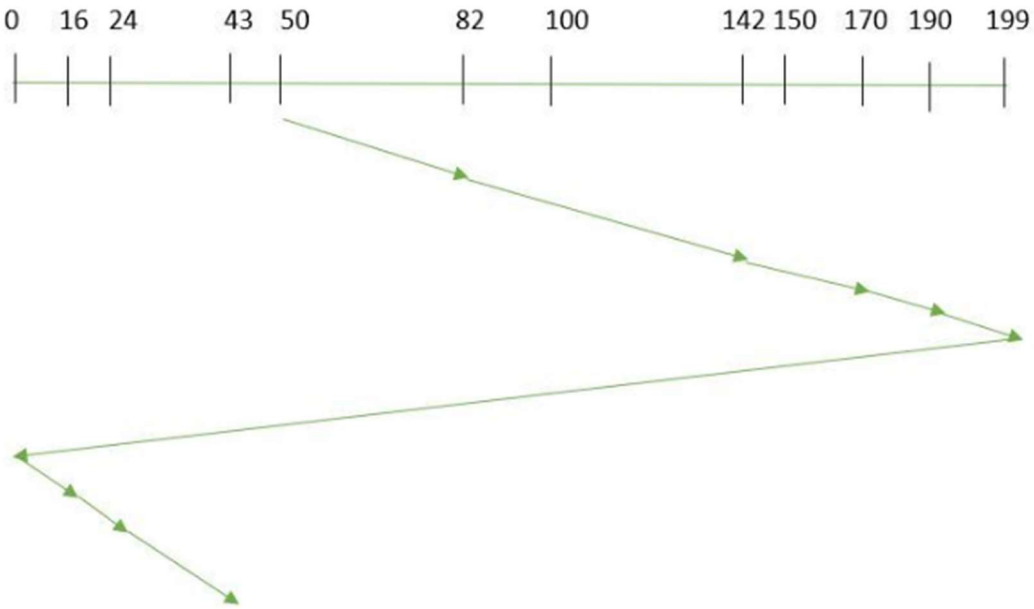
4) C-SCAN (Circular SCAN)

Explanation:
C-SCAN is a modified version of SCAN. The disk arm moves in **one direction only**. After reaching the end, it **returns to the beginning without servicing requests** on the return.

Working:

- Services requests in one direction.
- Jumps back to the start and repeats.

Example:



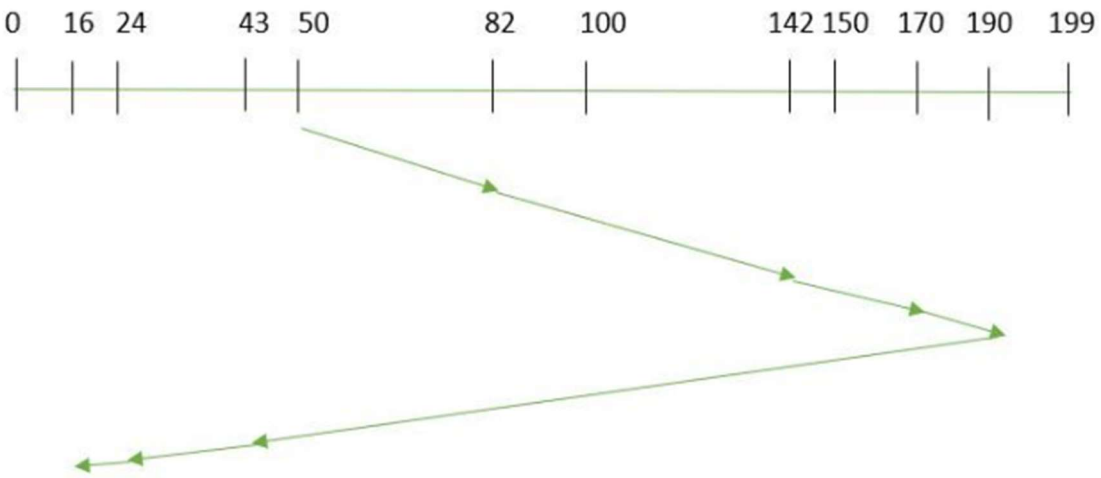
Suppose the requests to be addressed are- 82,170,43,140,24,16,190 and the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**
So, the total overhead movement (total distance covered by the disk arm) is calculated as:
=(199-50) + (199-0) + (43-0) = 391

5) LOOK Algorithm

Explanation:
LOOK is similar to SCAN, but the disk arm **does not go to the extreme ends** unless there is a request.

Working:

- Disk arm moves only up to the last request in one direction.
- Then reverses direction.



Suppose the requests to be addressed are- 82,170,43,140,24,16,190 and the Read/Write arm is at 50, and it is also given that the disk arm should move "towards the larger value".

So, the total overhead movement (total distance covered by the disk arm) is calculated as:
= (190-50) + (190-16) = 314

(c) Explain the Banker's algorithm for deadlock avoidance with an example.

(c) Explain the Banker's Algorithm for Deadlock Avoidance with an Example

---

What is Banker's Algorithm?

The **Banker's Algorithm** is a **deadlock avoidance algorithm** used in operating systems. It ensures that the system **never enters an unsafe state** by carefully checking whether granting a resource request will keep the system in a **safe state**.

The algorithm is named *Banker's* because it works like a banker who lends money only if he can be sure that all customers can eventually repay their loans.

Basic Concepts Used in Banker's Algorithm

1. **Available**
   - A vector showing the number of available instances of each resource type.
2. **Max**

    - Maximum number of resources each process may need.
3. **Allocation**
    - Resources currently allocated to each process.
4. **Need**
    - Remaining resources required by each process.
    - **Need = Max − Allocation**

Safe State

A system is said to be in a **safe state** if there exists at least one **safe sequence** of processes such that every process can complete execution using available resources plus resources released by previously completed processes.

Steps of Banker's Algorithm

1. When a process requests resources, check if:
    - Request ≤ Need
    - Request ≤ Available
2. If yes, **pretend to allocate** the resources.
3. Run the **safety algorithm** to check:
    - If the system is still in a safe state → grant the request.
    - If not → deny the request.

Safety Algorithm

1. Initialize:
    - Work = Available
    - Finish[i] = false for all processes
2. Find a process such that:
    - Finish[i] = false
    - Need[i] ≤ Work
3. If found:
    - Work = Work + Allocation[i]
    - Finish[i] = true
4. Repeat until:
    - All Finish[i] = true → system is safe
    - No such process exists → system is unsafe

Example of Banker's Algorithm

Given:

- Number of processes = 5 (P0 to P4)
- Number of resource types = 3 (A, B, C)

Available Resources

A	B	C
3	3	2

Allocation Matrix

Process	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max Matrix

Process	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Need Matrix (Max − Allocation)

Process	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Safety Check

- Work = (3, 3, 2)

Possible safe sequence:

1. **P1** → Need (1,2,2) ≤ Work
    - Work = (3,3,2) + (2,0,0) = (5,3,2)
2. **P3** → Need (0,1,1) ≤ Work
    - Work = (5,3,2) + (2,1,1) = (7,4,3)
3. **P4** → Need (4,3,1) ≤ Work
    - Work = (7,4,3) + (0,0,2) = (7,4,5)
4. **P0** → Need (7,4,3) ≤ Work
    - Work = (7,4,5) + (0,1,0) = (7,5,5)
5. **P2** → Need (6,0,0) ≤ Work
    - Work = (7,5,5) + (3,0,2) = (10,5,7)

✅ **Safe Sequence:**
P1 → P3 → P4 → P0 → P2
